VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

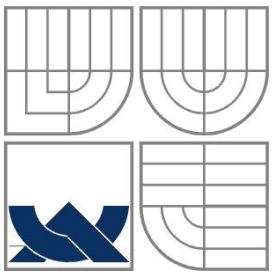
DATOVÉ STRUKTURY S PARALELNÍM PŘÍSTUPEM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

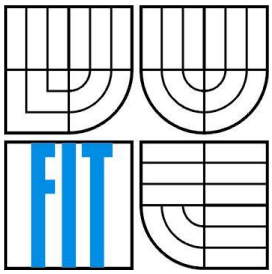
AUTOR PRÁCE
AUTHOR

TOMÁŠ OPLETAL

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

DATOVÉ STRUKTURY S PARALELNÍM PŘÍSTUPEM

DATA STRUCTURES WITH PARALELL ACCESS

BAKALÁŘSKÁ PRÁCE
BACHELOR‘S THESIS

AUTOR PRÁCE
AUTHOR

TOMÁŠ OPLETAL

VEDOUCÍ PRÁCE
SUPERVISOR

Mgr. LUKÁŠ HOLÍK, Ph.D.

BRNO 2014

Abstrakt

Paralelní programování přináší, kromě možnosti rozložit běh programu na více současně běžících procesů sdílejících data, také některé problémy. Je potřeba tyto paralelně běžící procesy synchronizovat a zařídit, že při komunikaci a sdílení dat nedojde k problémům vycházejícím z toho, že běží mnoho procesů současně. Tyto synchronizační algoritmy také nesmí příliš zatížit celkový běh programu. Tato práce popisuje způsoby synchronizace procesů a také je zde implementováno několik různých algoritmů pro práci s paralelní frontou, které jsou výkonnostně otestovány.

Abstract

Parallel programming brings out, apart from the opportunity to spread out a program execution to many simultaneously running processes sharing data, some new problems. It is necessary to synchronize these processes running in parallel and make sure that during the process communication and data sharing there will not arise any troubles. These synchronization algorithms also cannot use too much resources that are otherwise used for the actual program.
This thesis describes ways of process synchronization and also provides an implementation of several algorithms for parallel queue. Implemented algorithms was also tested for their performance.

Klíčová slova

Paralelní programování, datová struktura, algoritmus, měření výkonnosti

Keywords

Parallel programming, data structure, algorithm, performance testing

Citace

Opletal Tomáš: Datové struktury s paralelním přístupem, bakalářská práce, Brno, FIT VUT v Brně, 2014

Datové struktury s paralelním přístupem

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Lukáše Holíka, Ph.D.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Tomáš Opletal
20. 5. 2014

Poděkování

Rád bych poděkoval vedoucímu práce Ing. Lukáši Holíkovi, Ph.D. za poskytnuté rady a materiály pro tvorbu této práce. Také bych chtěl poděkovat Ing. Zdeňku Letkovi, Ph.D., který mi umožnil otestovat celou aplikaci na multiprocesorových systémech.

Obsah

Kapitola 1

Úvod

Procesory byly, jsou a zřejmě ještě dlouhou dobu budou nejdůležitějšími prvky počítače. Dříve bylo snahou firem, zabývajících se návrhem procesorů, vytvořit procesor s co největší taktovací frekvencí.

Čím více však frekvence rostly, tím více se projevoval jeden z největších problémů, kterými se snaží tyto firmy také zabývat – nutnost dostatečného chlazení. Na procesory s velmi vysokou frekvencí prostě přestávalo konvenční chlazení stačit.

Současný trend ve vývoji procesorů se tak ubírá jiným směrem – nejedná se již o neustálé zvyšování frekvence, ale o zvyšování počtu jader na procesoru. Každé jádro na procesoru může pracovat samostatně a tím tedy výrazně zvýšit výkon systému, a to i přes zachování nižších frekvencí, které lze bez problému uchladit. Dnes tak již téměř každý notebook či dokonce i mobilní telefon obsahuje procesor, který má alespoň dvě jádra. Programování aplikací, které jsou schopny paralelního běhu, se nazývá paralelní programování. Paralelní programování však existovalo i dávno předtím. I na jednojádrovém procesoru lze spustit několik vláken současně. Vícejádrové systémy ale přinášejí skutečnou hardwarovou paralelizaci.

Používání paralelizace však s sebou přináší nové problémy – například může nastat situace, kdy dvě jádra začnou zapisovat v jeden okamžik do stejného místa v paměti. Většina těchto problémů je známá už delší dobu a byly také navrženy různé mechanismy pro jejich řešení.

V kapitole č. 2 jsou popsány základní techniky, které se používají v paralelních systémech, jež umožnují bezproblémový běh více procesů současně. Jsou zde zmíněny základní problémy vznikající při práci s paralelními systémy a také různé způsoby jejich řešení. Zabývám se zde jak blokujícími, tak neblokujícími způsoby synchronizace.

Kapitola č. 3 popisuje základní datové struktury a jejich pozici a nová úskalí při použití v paralelním systému. Je zde popsáno na co je potřeba se zaměřit při návrhu algoritmů pro datové struktury a také způsob testování paralelních algoritmů.

Kapitola č. 4 pak popisuje fungování a implementaci celé aplikace, dále obsahuje popis implementovaných algoritmů, a také představí výsledky testů a zhodnocení výsledků těchto testů.

Kapitola 2

Sdílená paměť

Sdílená paměť představuje způsob, jak lze datovou strukturu sdílet mezi procesy bez vytváření redundantních kopií všech dat, které chceme sdílet a bez nutnosti vytvářet nějaký speciální protokol pro komunikaci mezi procesy na zasílání zpráv o výsledcích a průběhu operace. Pod pojmem sdílená paměť si můžeme často představit blok RAM paměti, k němuž přistupují jádra procesorů vykonávající program. Tento model však s sebou přináší některé nevýhody – komunikace procesoru s pamětí je obecně pomalá a při přístupu několika procesorů ke sdílené paměti se toto projeví ještě výrazněji. Obyčejně se problém rychlosti přístupu do paměti řeší cachováním, kdy si procesor používaná data uloží do vlastní rychlé paměti cache. Zde při paralelním běhu programu vzniká nový problém – je nutno zajistit, aby vždy, když procesor nějak modifikuje data uložená ve své paměti cache a třeba je i uloží zpátky do hlavní paměti, se tyto změny promítly i do dat uložených v cache ostatních procesorů, protože tyto procesory mají ve své cache paměti uloženou starou verzi dat. Náročnost tohoto úkolu se pak ještě zvyšuje s narůstajícím počtem úrovní paměti cache. Dnešní běžné procesory obyčejně mívají L1, L2 a L3 cache, tedy 3 různé úrovně paměti cache. Paralelní systémy kde je zajištěno, že nevznikne problém s manipulací zastaralých dat v cache, se nazývají cache-koherentní systémy. Garantují, že v určitém časovém úseku se všechny procesory dozví o změnách v nacachovaných datech a tyto změny stejného paměťového místa se projeví ve stejném pořadí všem běžícím vláknům.

Existují dvě základní architektury sdílené paměti – UMA a NUMA. UMA, neboli Uniform Memory Access, znamená, že všechny procesory sdílejí paměť rovnoměrně. Přístupová doba do paměti je u všech procesorů stejná a nezáleží, kde se dané paměťové místo nachází. Opakem je NUMA – Non-Uniform Memory Access, zde přístupová doba do paměti záleží na skutečném fyzickém umístění

procesoru a umístění paměti a každý procesor má většinou svou lokální paměť, do které je přístup nejrychlejší.

Systémy kde je paměť fyzicky oddělená a lze je adresovat jako jeden paměťový prostor, se nazývá distribuovaná sdílená paměť.

Zasílání zpráv

Zasílání zpráv představuje jiný způsob komunikace mezi dvěma objekty – v paralelním prostředí se jedná o výměnu informací mezi procesy nebo vlákny. Paralelní procesy si vyměňují data a informace tím, že si posílají mezi sebou zprávy. Posílání zpráv se může dít asynchronně nebo synchronně.

2.1 Synchronizace

Kvůli způsobu, jakým dnes pracují počítače a zvláště procesory, je nutné zamezit případům inkonzistence, tedy případům, kdy např. jeden program čte data ze struktury, zatímco jiný program do této struktury simultánně zapisuje. Úsek programu, jenž přistupuje ke sdílené datové struktuře, a kde mohou nastat tyto případy inkonzistence, se nazývá kritická sekce.

Standardní přístup, jak zamezit inkonzistenci při provádění příkazů v kritické sekci, je použití takzvaného vzájemného vyloučení (angl. mutual exclusion). Díky vzájemnému vyloučení se potom vždy jen jeden proces nachází v kritické sekci a manipuluje s datovou strukturou.

2.1.1 Blokující mechanismy

Mechanismy popsané v následujících odstavcích jsou tzv. blokující. Poskytují ochranu kritické sekce tak, že vždy umožní vstup pouze jednomu procesu. Ostatní procesy, které se dostanou ke kritické sekci a chtějí do ní vstoupit, musí čekat na uvolnění kritické sekce procesem, který se v ní právě nachází. Tímto se však prodlužuje doba vykonávání programu. Procesy mrhají cenný čas čekáním, místo aby prováděly výpočty.

Použití blokujících mechanismů pro implementaci ochrany kritické sekce s sebou nese také mnohá úskalí. Při nepromyšleném použití tento způsob může výrazně omezit běh procesů, protože procesy mohou čekat na uvolnění jednoho či několika zámků, například okolo celé datové struktury. Mohou se také objevit kritické problémy, které by způsobily pád či zablokování celé aplikace. Tyto problémy jsou popsány v oddíle 2.1.2.

Základní blokující mechanismy, jež se používají u paralelních algoritmů pro ostrahu kritické sekce, jsou tzv. zámky (angl. locks), dále semfory (semaphores) a monitory.

Semafory

Semafor je synchronizační prostředek, jenž obsahuje čítač. Tento čítač funguje jako počítadlo, které říká, kolik zbývá ještě volných prostředků. Hodnota čítače je inicializována na hodnotu volných prostředků. Semafory byly vymyšleny nizozemským matematikem E. Dijkstrou [6]. Pro obsluhu semaforů existují dvě operace – V (inkrementace čítače) a P (dekrementace/čekání). Operace P dekrementuje čítač volných prostředků a pokud je tento čítač menší než 0, pak proces, jenž operaci P prováděl, je blokován a přejde do fronty čekajících procesů. Operace V potom čítač inkrementuje a pokud je inkrementovaná hodnota větší nebo rovna 0, znamená to, že jsou ještě ve frontě čekající procesy. Vyjme tedy proces z fronty čekajících procesů a připraví jej na pokračování v běhu.

Semafory, jejichž čítač se může nacházet jen ve stavu 1 a 0, se nazývají binární semafory.

Semafory pouze značí, kolik je volných prostředků a ne nutně jaké prostředky jsou volné, a které procesy si vyžádaly tyto prostředky. Bezpečnost z hlediska nevzniknutí deadlocku (popsán v oddílu 2.1.2) nebo neuvolnění semaforu cizím procesem nelze zaručit.

Zámky

Zámky jsou podobné binárním semaforům, avšak s tím rozdílem, že jeden zámek vždy dostane jeden proces a pouze ten jej potom může uvolnit. Zámky tedy celou datovou strukturu nebo její část uzamknou a tím povolí přístup k ní pouze procesu,

který si tento zámek vyžádal a ostatním procesům přístup k datové struktuře na čas zamezí. Zámky poskytují dvě základní operace – pokus o získání (zamknutí) zámku, a pokud je zámek zamknut, tak čekání na uvolnění. Druhou operací je odemknutí zámku procesem, jenž daný zámek zamknul. Zámky poskytují výhody oproti semaforům, primárně tedy tím, že zámek nemůže odemknout cizí proces. Jelikož zámek má svého vlastníka, vzniká také možnost řešit inverzi priorit (detailně popsáno v oddílu 2.1.2), pokud dojde ke stavu, že na zámku začne čekat proces s vyšší prioritou, než má současný proces, který se nachází v uzamknuté kritické sekci.

Existují různé druhy zámků, každý je vhodný na jiný typ úlohy a záleží také na systému, na kterém bude program běžet.

Základním typem zámku je mutex. Mutex je nejjednodušší typ zámku, implementuje základní funkcionalitu zámků.

Často používaným typem zámku je tzv. spin zámek (spin-lock). Jedná se o druh zámku, který nutí proces stále běžet ve smyčce a dotazovat se, zda již byl zámek uvolněn. Proces tak neustále spotřebovává procesorový čas.
Spin-zámky jsou vhodné pro použití hlavně tehdy, pokud očekáváme, že zámek bude vždy zamčen jednotlivými procesy pouze na krátkou dobu (nejlépe, aby nedošlo vůbec k přepnutí kontextu). Čím déle bude zámek držen, tím vícekrát dojde k přepnutí kontextu a ostatní čekající procesy budou jen spotřebovávat procesorový čas neustálým dotazováním se na status zámku.

Jiným typem zámku je readers-writer zámek, který při operaci čtení z datové struktury připouští přístup více procesů, ale při operaci zápis povolí přístup pouze jednomu procesu. Zde může dojít k problému vyhladovění písařů, kdy při vysoké vytíženosti datové struktury mohou stále do kritické sekce vstupovat noví a noví čtenáři.

Existuje také zámek pro rekurzivní použití – reentrant mutex neboli rekurzivní zámek. Ty povolují procesu, který zámek původně zamknul, opakované zamykání - např. právě při používání rekurzivních funkcí. K odemčení rekurzivního zámku dojde až tehdy, pokud byl odemčen právě tolikrát, kolikrát byl i zamknut procesem, jež zámek vlastní.

Monitory

Monitor je vysokoúrovňový prostředek pro zajištění vzájemného vyloučení. Většinou bývají monitory implementovány přímo jako součást programovacího jazyka jako zvláštní třída. Jako prostředek pro vynucení vzájemného vyloučení často používá nějaký zámek (např. mutex) a doplňuje další funkcionalitu – např. je možné přidat další podmínky, které musí být splněny pro vstup do kritické sekce. Monitory jsou snadněji použitelné pro programátora a při jeho použití je menší riziko vzniku chyby.

2.1.2 Problémy blokujících mechanismů

Používání blokujících algoritmů s sebou přináší i problémy, některé z nich mohou významně ovlivnit celý běh aplikace. Většina problémů souvisí s čekáním několika procesů na uvolnění zámku. Hlavní problémy, jež se mohou vyskytnout v programech používajících blokující mechanismy:

Jsou blokující

Pokud některý proces zamkne zámek, další procesy, které chtějí pokračovat a jsou schopny běžet, jsou zbytečně zastaveny při čekání na zámek. Doba vykonávání programu se tak prodlužuje.

Od tohoto se odvíjí problém, zvaný **Convoying** – několik procesů se stejnou prioritou, zhruba ve stejném čase, zažádají o zámek. Zámek získá pomalý proces a ostatní procesy se tak zpomalí na jeho rychlost.

Dále například uvažme situaci, kdy proces vlastní zámek a nachází se v kritické sekci. Tento proces je najednou ukončen (např. pomocí příkazu *kill*), či zhavaruje. Co se stane dále?

Deadlock

Při nesprávném použití zámků se mohou procesy dostat do situace, jež se nazývá deadlock (česky uváznutí). Při této situaci procesy čekají na uvolnění zámku, jež je vlastněn procesem, který nemůže pokračovat v činnosti, protože čeká na

uvolnění zámku, jež vlastní nějaký z čekajících procesů. K uváznutí může v systému dojít pouze, pokud jsou splněny tzv. Coffmanovy podmínky. Pokud jsou všechny tyto podmínky splněny, pak může dojít k uváznutí.

Coffmanovy podmínky jsou následující [1]:

1. Mutual exclusion (vzájemné vyloučení) - V systému existuje nutnost vzájemného vyloučení – existují sdílená data, jež může používat jen jeden proces v jednom okamžiku
2. Hold and Wait (drž a čekej) - Proces, jenž má přiděleny nějaké prostředky, může zažádat o další.
3. No preemption (neodnímatelnost) – Proces se musí sám vzdát přiděleného prostředku, nelze mu jej odebrat.
4. Circular wait (čekání do kruhu) – Existuje cyklus procesů, čekajících na prostředky, kde je nějaký prostředek přidělen jednomu čekajícímu procesu a tento prostředek je vlastněný jiným procesem, který čeká na prostředek vlastněn prvním procesem.

Odstranění kterékoliv z Coffmanových podmínek zaručuje odstranění uváznutí, v praxi je toto však většinou velmi obtížné a někdy neproveditelné.

Inverze priorit

Situace, jenž nastává, pokud méně prioritní proces získá zámek, a začne provádět operace v kritické sekci těsně před vysoko prioritní procesem, jež chce přes zámek vstoupit do kritické sekce, ale musí čekat na nízko prioritní proces. Navíc mezitím nějaký středně prioritní proces, který nečeká na kritické sekci, může začít běžet, a tím ještě více zdržet čekající vysoko prioritní proces, protože má větší prioritu než nízko prioritní proces, jež je v kritické sekci. Tomu pak trvá příliš dlouho opustit kritickou sekci, protože procesor je zaneprázdněn vykonáváním procesu se střední prioritou.

Existují různé mechanismy, které se snaží inverzi priorit řešit. Zmiňme například Priority inheritance (dědění priorit) – nízko prioritní proces zdědí po

dobu běhu v kritické sekci prioritu vysoko prioritního procesu, a tím nemůže dojít k tomu, že středně prioritní proces, který může běžet jinde, bude zdržovat svým během nízko prioritní proces v kritické sekci.

Dalším podobným řešením je priority ceiling (strop priorit) – proces, jež má na starosti režii a obsluhu zámku, má nastavenu vysokou prioritu, která je předána procesu, jež si zámek vyžádal. Ostatní procesy ale nesmí mít prioritu větší, než má onen proces, který obsluhuje zámek.

Celkově je na zámky také potřeba vynaložit nemalé systémové zdroje, např. procesorový čas pro zamykání a odemykání zámků a pro samotnou inicializaci a destrukci zámků. Obecně zámky omezují škálovatelnost celého programu a programy naopak nabývají na složitosti.

Následující text popisuje vlastnosti algoritmů, jež určují, zda může v algoritmu dojít např. k uváznutí nebo zda algoritmus skončí v konečném počtu kroků. Dokazování těchto vlastností je nedílnou součástí nově prezentovaných paralelních algoritmů.

Safety – bezpečnost

Bezpečnost říká, že se nikdy nestane nic špatného při běhu algoritmu. U paralelních algoritmů například chceme garantovat, že nedojde k uváznutí. Tedy by mělo být zcela vyloučeno, že dojde k uváznutí, i když jsou jednotlivé instrukce jednotlivých vláken prováděny jakkoliv. Uváznutí se dá detekovat pomocí výše popsaných Coffmanových podmínek. Další vlastností, kterou chceme u paralelního algoritmu garantovat, může být například atomicita – chceme, aby jednotlivé operace vypadaly, jako že proběhly atomicky. Atomicita je blíže popsána v oddílu 2.1.3. Bezpečnostní podmínky jsou uspokojeny v nekonečném čase.

Liveness

Garantuje, že eventuálně musí dojít k něčemu „dobrému“. Tyto podmínky jsou uspokojeny v reálném čase. Chceme garantovat, že všechny procesy eventuálně skončí a případně vrátí návratovou hodnotu. Příkladem může být podmínka, že se v kritické sekci nenachází žádná možnost, jak by se algoritmus mohl dostat do nekonečné smyčky a zablokovat tak pokrok nejen sobě, ale i všem ostatním procesům, čekajícím na vstup do kritické sekce.

2.1.3 Základní synchronizační atomická primitiva

Atomicita

Pokud je operace (nebo množina operací) atomická, znamená to, že operace vypadá, jako by se provedla celá v jeden okamžik. V anglické literatuře často bývá atomicita nazývána jako linearizability. Poprvé byla popsána v Herlihy and Wing [7]. Atomicita je pro paralelní programování zcela základní koncept, jelikož umožnuje použití sdílených objektů bez jakýchkoliv dalších zdrojů. Pokud navíc máme celý objekt sestávající se ze dvou a více atomických struktur - například atomických registrů (jejich operace read a write jsou prováděny atomicky), lze na celý tento objekt pohlížet jako na atomický objekt, protože jednotlivé operace i v paralelním systému budou vypadat, jako kdyby byly prováděny sekvenčně jedna za druhou, čili jako kdyby byla vykonávána jen jedna operace v daném okamžiku [2].

Forma implementace atomicity je např. použití zámků pro vynucení vzájemného vyloučení. Operace prováděné v kritické sekci pak tedy vypadají jako atomické.

Atomicita samotná je neblokující. Nevyžaduje totiž čekání na dokončení jiných operací, protože jednotlivé operace vypadají, jako kdyby proběhly v jeden okamžik najednou. Blokování tak může být pouze vlastnost nějaké konkrétní implementace, která atomicitu zajišťuje.

Pro zajištění konzistence paralelních operací musí systém, na kterém operace běží, poskytnout minimální atomické operace, bez kterých se žádná implementace paralelního algoritmu neobejde. Jedná se v podstatě o operace čtení/zápis do paměťového místa, jež je provedena atomicky.

K implementaci těchto atomických operací se využívají instrukce procesoru, jež zajišťují atomicitu na hardwarové úrovni. Samotné atomické operace pak mohou být součástí překladače, operačního systému nebo jako knihovny ve vyšších programovacích jazycích. Některé procesory již poskytují tyto atomické operace přímo jako svou instrukci. Různé systémy a jazyky poskytují různé varianty atomických operací.

Mezi základní atomické operace, jež manipulují s jednou proměnnou, patří Test-And-Set (TAS), Fetch-And-Add (FAA) a Compare-And-Swap (CAS).

Compare-and-swap má 3 parametry – paměťovou lokaci, starou hodnotu, jež je na tomto místě očekávána, a novou hodnotu, která má být na toto místo zapsána. Nejčastěji se CAS implementuje jako operace s návratovou hodnotou typu boolean, jež indikuje, zda se zapsání nové hodnoty podařilo provést v pořádku či nikoliv. Druhá implementace vrací hodnotu, jež byla přečtena z daného paměťového místa před zápisem nové hodnoty. Compare-and-swap se používá pro implementaci např. semaforů či různých typů zámků. Je také velmi hojné používán v bezzámkových algoritmech. Algoritmus compare-and-swap, který vrací boolean je zobrazen v algoritmu 1.1

```
bool compare_and_swap(int *reg, int oldval, int newval)
{
    if ( *reg  == oldval ) // je hodnota v paměti stále nezměněná?
    {
        *reg = newval;     // zapsání nové hodnoty do registru
        return true;
    }
    else                   // hodnota v paměti se změnila
        return false;
}
```

Algoritmus 1.1: Compare-and-swap

Test-and-set operace má pouze jeden parametr – paměťovou lokaci. Do této paměťové lokace pak atomicky zapisuje, nejčastěji hodnotu 1. Návratovou hodnotou je hodnota paměťového místa před zápisem. Test-and-set se často používá pro implementaci zámků – hlavně zámku typu spinlock. Test-and-set se neustále testuje ve smyčce a snaží se uzamknout semafor. Pokud se stále vrací návratová hodnota 1 (či jiná konstanta odpovídající zamknutému semaforu), stále se nacházíme ve smyčce. V okamžiku, kdy test-and-set vrátí 0, znamená to, že semafor byl odemčen a čekání ve smyčce se přeruší a proces může vstoupit do kritické sekce.

Fetch-and-add operace bere jako parametr také pouze paměťovou lokaci. Hodnotu na tomto paměťovém místě zvětší o 1. Návratová hodnota je hodnota paměťového místa před provedením změny. Celá funkce musí být provedena atomicky. Tato operace se také používá k implementaci nástrojů vynucujících vzájemné vyloučení, jako jsou semafory apod. Existují i jiné další varianty této operace, lišící se v prováděné operaci – místo přičítání se může jednat o odečítání či operaci nand apod.

ABA problém

ABA problém vzniká tehdy, pokud je stejné místo v paměti přečteno dvakrát, pokaždé má stejnou hodnotu, avšak mezitím již bylo toto paměťové místo modifikováno a potom zase navráceno nazpět na předchozí hodnotu. Zde může dojít i při čtení stejné hodnoty k neočekávanému chování, a to díky skryté modifikaci jiným procesem.

Příkladem může být přečtení, odstranění a vložení nového prvku do pole. Tento nový prvek bude často umístěn do stejného místa v paměti, kde byl i starý prvek. Ukazatel na starý prvek je potom totožný s ukazatelem na nový prvek.
ABA problém postihuje například synchronizační operaci compare and swap.

Řešení ABA problému

Jedním z možných řešení je přidat čítač modifikací k danému paměťovému místu. Například se vyhradí několik bitů adresy pro tento čítač. Potom i přes přečtení stejné hodnoty v paměti nebude odpovídat hodnota v čítači modifikací a proces pozná, že na daném paměťovém místě proběhly změny a musí číst znovu.

Jiné řešení využívá operace LL/SC (load-linked, store-conditional), která umožní změnu paměťového místa pouze za předpokladu, že dané paměťové místo nebylo změněno během provádění páru LL + SC. V současné době LL/SC není příliš podporován procesory a pokud ano, jeho implementace je slabá, tj. LL/SC může proběhnout neúspěšně i pokud došlo k paměťovým operacím někde úplně jinde, či z jiných neočekáváných důvodů.

Předchozí atomická primitiva vždy manipulují pouze s jednou proměnnou. Existují také primitiva pro 2 a více proměnných, nejznámější algoritmus je Double compare-and-swap (DCAS). Je nutno uvést, že tento algoritmus pracuje s dvěma různými proměnnými na různých paměťových místech, nikoliv, jak by se mohlo na první pohled zdát, s proměnnou, jež má velikost double-word (často 64bit). Podpora double compare-and-swap v hardwaru je však velmi malá, na některých procesorech byla úplně zastavena kvůli velké časové náročnosti.

2.1.4 Neblokující algoritmy

Tradiční přístup, použití blokujících mechanismů pro vynucení vzájemného vyloučení, není kvůli výše popsaným problémům příliš žádaný. Moderní přístup, který se začíná prosazovat v oblasti paralelních algoritmů pro zamezení inkonzistence, je použití tzv. neblokujících algoritmů.

Neblokující algoritmy jsou algoritmy, jež zámky a jiné blokující mechanismy nepoužívají při zachování konzistence v paralelních systémech. Snaží se problémům, vznikajícím při použití zámků, zamezit a výrazně tak zjednodušit a zrychlit běh programu. U těchto algoritmů nemůže například dojít k deadlocku a jiným problémům, které mohou postihovat zámky. Nejčastěji využívají tyto algoritmy právě základních atomických primitiv pro zajištění minimální nutné

režie potřebné k zachování konzistence sdílených datových struktur. Existují však také algoritmy, jež využívají pouze operace atomického čtení nebo zápisu. Neblokující algoritmy nebrání procesům k přístupu ke sdílené datové struktuře a pouze při přímém konfliktu mezi operacemi procesů se operace neprovedou a procesy ji zkusí opakovat.

V současnosti se neblokující implementace algoritmů prosazují v různých knihovnách i jako součásti samotných programovacích jazyků.

Dle síly neblokujícího algoritmu se rozlišují následující typy:

Obstruction-free

Tyto algoritmy jsou nejslabší mezi neblokujícími algoritmy. Obstruction-free algoritmy garantují, že proces běžící v izolaci někdy skončí v konečném počtu kroků. Pokud proces běží v izolaci, znamená to, že v daném okamžiku žádné jiné operace či procesy nepracují s daným objektem. Toto může znamenat, že jiné operace či procesy jsou započaty, ale v současné době jsou pozastaveny.

Lock-free

Lock-free algoritmy jsou „slabšími“ verzemi neblokujících algoritmů. Tyto algoritmy garantují celkový postup, může však nastat vyhladovění, tj. některé procesy stále mohou bez úspěchu opakovat přístup ke sdílené datové struktuře do nekonečna. Garantují tedy nějaký postup alespoň jednoho procesu. Řeší problémy použití zámků – nemůže nastat deadlock, či inverze priorit. Všechny lock-free algoritmy jsou i obstruction-free.

Wait-free

Wait-free jsou vylepšením lock-free algoritmů. Garantují, že všechny procesy za určitou časovou jednotku budou moci udělat nějaký pokrok a eventuálně skončit v konečném počtu kroků. Garantují, že nemůže nastat vyhladovění některých procesů. Tyto algoritmy většinou bývají řádově náročnější než lock-free algoritmy.

Wait-free verze algoritmů se příliš nevyskytují jak v praxi, tak v teoretickém výzkumu. Všechny wait-free algoritmy jsou také lock-free.

Consensus

Consensus number vyjadřuje hodnotu, pro kolik maximálně současně běžících paralelních procesů může objekt vyřešit tzv. consensus problém. Ten vyjadřuje, že procesy, které chtějí naráz zapisovat na stejné místo, se musí shodnout na jedné základní hodnotě. Procesy musí přednést své hodnoty, shodnout se na jedné a toto všechno vykomunikovat s ostatními. Přitom také nesmí dojít k pádu či nedefinovanému chování. Tedy popisuje, pro kolik procesů je garantován wait-free postup. Algoritmy s nekonečným consensus number jsou univerzální – mohou být použity k wait-free implementaci jakékoliv datové struktury. V tabulce 1.1 jsou některé základní operace používané v paralelním programování a jejich consensus number [10].

Atomické primitivum	**Consensus number**
čtení/zápis do registru	1
Test-and-set	2
Fetch-and-add	2
Compare-and-swap	∞
LL/SC(load-linked, store-conditional)	∞

Tabulka 1.1: Consensus number pro základní operace

Korektnost

U jednoduchých algoritmů, jež využívají zámky, bývá většinou jednoduché ověřit, že se algoritmus chová správně. Povětšinou bude algoritmus stejný jako pro stejnou datovou strukturu v sekvenčním prostředí. Obecně však, zvláště pro neblokující algoritmy, toto nebývá jednoduché ověřit. Ověření je však nutností, je

potřeba zajistit, aby se program ve skutečnosti choval tak, jak to je zamýšleno při návrhu algoritmu.

Algoritmy pro sekvenční datové struktury mají jasně daný a předvídatelný pořádek operací, jednotlivé operace jdou za sebou po jednom, pouze jedna běžící operace v daném čase. Tento popis se však nevztahuje na paralelně běžící procesy. Jednotlivé operace nejsou přesně vymezeny v pořadí. Pro ověřování správnosti chování algoritmu existují tzv. podmínky korektnosti.

Jedna z těchto podmínek korektnosti se nazývá atomicita (linearizability), popsána v oddílu 2.1.3

Kapitola 3

Datové struktury s paralelním přístupem

Datové struktury obecně poskytují programátorovi způsob, jak uložit data tak, aby práce s nimi byla efektivní a relativně snadná a také poskytují algoritmy pro práci s těmito daty. Výběr správné datové struktury je zcela jistě jeden z klíčových prvků při návrhu a implementaci programu. Zvolení nevhodné datové struktury může celý program několikanásobně zpomalit. Stejně tak důležitým faktorem je i efektivita a další vlastnosti algoritmů, jež bude daná datová struktura používat.

Algoritmy, použité v programu, by měly odrážet jak požadavky na program a jeho využití, tak například i hardware, na kterém se bude daný program provozovat.

Paralelní programování přináší možnost, jak mohou nezávislé programy sdílet různé zdroje, kupříkladu právě datové struktury uchovávající sdílená data. V paralelním programování, kde se předpokládá zvýšená doba práce s daty, tak důležitost zvolení správného algoritmu a jeho implementace jen narůstá.

Velkou snahou programátorů zabývajících se algoritmy pro paralelní datové struktury je omezit celkovou režii související s přístupem k datové struktuře s paralelním přístupem na minimum tak, aby co nejméně zatěžovaly výpočetní výkon a program se mohl soustředit na jeho skutečný úkol.

3.1 Základní datové struktury podporující paralelní přístup

Register

Základní datová struktura, která umožňuje sdílet jednoduchou hodnotu mezi mnoha písaři a čtenáři.

Systém sdílené paměti většinou poskytuje základní operace čtení/zápis atomicky.

Kolekce

Kolekce je základní datová struktura, která obsahuje prvky obecně stejného typu a je neuspořádaná.

Základní operace: Add (vložit prvek)

Retrieve (vyjmout prvek)

Pole

Základní datová struktura, jež je často využívána k implementaci dalších, více pokročilých datových struktur. Prvky pole jsou přístupné dle jejich indexu.

Základní operace: ReadAt (čtení prvku dle daného indexu)

WriteAt (zápis prvku na daný index)

Add (vložení prvku na konec pole)

Retrieve (čtení prvku z konce nebo začátku pole – dle implementace)

Lineární seznam

Dynamická datová struktura, jež se dá využít k implementaci několika abstraktních datových struktur. Skládá se z datových položek obecně stejného typu, které jsou lineárně provázány pomocí ukazatelů.

Existují seznamy jednosměrné a obousměrné. Prvky jednosměrného seznamu obsahují referenci jen na následující prvek. V obousměrném seznamu obsahují prvky kromě reference na následující prvek i referenci na prvek předchozí. Vždy jeden prvek seznamu je také označován jako aktivní a jeho pozici udává kurzor. Aktivní prvek označuje prvek, na kterém se právě nachází proces, který se seznamem pracuje.

Základní operace: InsertAfter / InsertBefore (vložení nového prvku za/před aktivní prvek)

Delete (smazání aktivního prvku)

Read (přečtení aktivního prvku)

Previous / Next (posun o jeden prvek zpátky / dále)

Zásobník

Zásobník je abstraktní datová struktura typu LIFO (last in, first out). Pro manipulaci s daty se udržuje odkaz na vrchol zásobníku – adresa poslední přidané položky (která také bude vyjmuta jako první při čtení ze zásobníku). Zásobník se používá především v sekvenčním prostředí, využití však najde i v paralelním systému.

Základní operace: Push (vložení prvku na vrchol zásobníku)
Top (čtení prvku z vrcholu zásobníku)
Pop (odstranění prvku z vrcholu zásobníku)

Fronta

Fronta je abstraktní datová struktura typu FIFO (first in, first out). Přístupný je vždy prvek, který byl do fronty přidán nejdříve. Fronty se sdíleným přístupem jsou zcela zásadní pro paralelní programování, využívají se například pro multiprocesorovou komunikaci a plánování nebo jako datový proud pro paralelní procesy. Fronta může být implementována například pomocí jednosměrného seznamu, kruhového pole aj.

Základní operace: Enqueue (vložení prvku na konec fronty)
Dequeue (výběr a odstranění prvku z čela fronty)

Double-ended queue (deque)

Double-ended fronta je datová struktura kombinující zásobník a frontu. Dostupné jsou vždy poslední a první prvek struktury.

Základní operace: PushLeft / PushRight (přidání prvku na začátek / konec)
PopLeft / PopRight (výběr prvku ze začátku / konce)

Prioritní fronta

Prioritní fronta je abstraktní datový typ, jenž obsahuje prvky i s informací o jejich prioritě. Při výběru z fronty se pak nejčastěji hledá prvek s nejvyšší prioritou. Toto lze velmi dobře využít např. při implementaci plánování založeném na prioritách procesů.

Základní operace: Find (vyhledat prvek s nejvyšší (nejnižší) prioritou)
Insert (vložit prvek do fronty)
Delete (odstranění prvku s nejvyšší (nejnižší) prioritou)

Asociativní pole

Každý prvek asociativního pole je dvojice klíč, hodnota. Každý z klíčů je unikátní v rámci pole. Každá hodnota má přirazen svůj klíč. Podle těchto klíčů se pak dá v poli vyhledávat a vkládat do něj. Tato struktura se nejčastěji implementuje pomocí hašovací tabulky.

Základní operace: Add (přidat dvojici klíč, hodnota)
Find (najít položku dle zadaného klíče)
Delete (smazat položku dle zadaného klíče)

Strom

Strom se skládá z kořene a poté z potenciálně mnoha uzlů - větví (uzly, mající další potomky) a větve jsou poté zakončeny listy (uzly bez potomka). Dle typu stromu mohou být uzly různě seřazeny či uspořádány. U stromu lze určovat zajímavé vlastnosti ovlivňující výkonnost stromu. Určuje se například výška stromu – nejdelší cesta od kořene k listu.

Základní operace: Insert (vložit uzel do stromu)
Delete (smazat uzel)
Find (vyhledání uzlu)

3.2 Neblokující datové struktury

Díky intenzivnímu vývoji a výzkumu neblokujících algoritmů a datových struktur lze dnes pro každý typ datové struktury najít mnoho neblokujících implementací, každá zaměřená k jinému využití a optimalizující jiné aspekty. Při výběru implementace je vždy třeba důkladně zvážit účely a využití datové struktury, jelikož často tyto implementace optimalizují nějaké aspekty na úkor jiných.

Mezi vlastnosti, na základě kterých se vybírá a optimalizuje konkrétní implementace, patří:

Časová složitost

Časová složitost je jedna z nejdůležitějších vlastností algoritmu. Určuje závislost času potřebného pro dokončení operace na velikosti vstupních dat.

Získává se analýzou algoritmu, obecně se rozlišuje časová složitost v nejlepším, nejhorším a průměrném případě. Nejčastěji se však omezujeme na určení časové složitosti v nejhorším případě.

V systémech s paralelně běžícími procesy je také velmi důležité rozlišit, zda maximální čas potřebný pro vykonání závisí na počtu současně běžících procesů.

Prostorová složitost

Prostorová složitost algoritmů popisuje využití paměti při běhu algoritmu. U některých algoritmů lze garantovat horní mez využití paměti, zatímco u jiných toto garantovat nelze a tyto algoritmy pak mohou časem teoreticky potřebovat neomezené množství paměti.

Škálovatelnost

Škálovatelnost neboli rozšiřitelnost označuje, zda je systém schopen v případně potřeby dostatečně zvýšit výkon tak, aby zvládl normálně pracovat. Synchronizační primitiva obecně nejsou škálovatelná.

Dynamická kapacita

V některých případech je důležité, aby datová struktura mohla dynamicky alokovat paměť dle potřeby. Pokud toto použitá datová struktura, potažmo algoritmus neumožňuje, kapacita datové struktury bude fixní od začátku po celou dobu života.

Konkurentní limit

Některé algoritmy, v závislosti na implementaci, nemusí povolit více než určitý počet konkurentních volání nějaké operace. Viz consensus number (oddíl 2.1.4)

Spolehlivost

Popisuje, jaké problémy se mohou vyskytnout u daného algoritmu (např. ABA problém, oddíl 2.1.3).

Kompatibilita

Kompatibilita označuje závislost algoritmu na platformě nebo programovacím jazyku.

Výše uvedené vlastnosti jdou většinou jasně určit, pokud běží program sekvenčně. Pokud však chceme zkoumat paralelní algoritmus, některé vlastnosti nebývá jednoduché určit – zejména časovou složitost. Jak se změní složitost algoritmu, jehož časová složitost je $O(n^3)$, pokud budou jednotlivé kroky běžet na n, n^2, n^3 procesorech? Umožnuje algoritmus vhodné rozčlenění jednotlivých kroků na mnoho procesorů? Podobné záležitosti se neřeší v sekvenčním prostředí, avšak při použití paralelního algoritmu je, jak ilustrováno výše, výpočet složitosti o mnoho složitější a závisí i na jiných faktorech, než je samotný kód algoritmu.

Pro popis paralelních výpočtů je potřeba definovat model, kterému bude možno zadat základní parametry (jako např. počet procesorů) a pomocí kterého půjde modelovat chování a běh paralelních algoritmů. Jelikož existuje mnoho způsobů, jakými mohou na paralelních systémech probíhat výpočty, jakými probíhá komunikace mezi procesy nebo existují různé způsoby sdílení dat a paměti, existuje také paralelních modelů celá řada. Existují procesorové modely, dále rozdělené na modely používající sdílenou paměť a modely se zasíláním zpráv jako formou komunikace. Také existují modely na úrovni obvodů, jež zachycují výpočty na velmi nízké úrovni komponent a komunikace.

3.3 PRAM

Parallel random-access machine (zkráceně PRAM) je abstraktní model systému se sdílenou pamětí. Byl vytvořen jako paralelní ekvivalent modelu RAM (random-access machine). PRAM je používán na modelování výkonnosti a chování paralelních algoritmů. PRAM si lze představit jako kolekci několika nezávislých

procesorů, které sdílí jednu centrální paměť. V tomto modelu jsou ignorovány některé základní vlastnosti počítačů reálného světa – jako například doba přístupu k různým druhům pamětí.

Tento přístup umožnuje modelu soustředit se přímo na samotnou výkonnost algoritmu, avšak na druhou stranu to může vést k některým nepřesnostem v simulaci, což může způsobovat zkreslená očekávání při skutečném nasazení. Existují však doplňky, jimiž lze některé tyto vlastnosti také simulovat.

Formální definice zní: PRAM se sestává z obecně neomezené množiny procesorů $P = \{P_0, P_1, \ldots\}$, kde každý procesor P_i je RAM procesor, neomezené centrální paměti, množiny vstupních registrů a konečného programu. Každý procesor má svou neomezenou paměť, akumulátor, programový čítač a označení, zda procesor v dané chvíli běží či nikoliv. Všechny procesory jsou identické a každý procesor P_i má schopnost rozeznat svůj vlastní index *i*. [2]
Libovolný procesor P_i může vždy v danou časovou jednotku buď přistoupit ke své lokální paměti, nebo ke sdílené globální paměti, nikdy ale nemůže přistoupit k lokální paměti jiného procesoru.

PRAM program je potom konečná množina instrukcí $\pi = (\pi_0, \pi_1, \ldots, \pi_m)$, a každá z těchto instrukcí je jednoho z typu LOAD, STORE, ADD, SUB, JUMP, JZERO, READ, FORK, HALT.

Při inicializaci modelu jsou vstupy vloženy do vstupních registrů, paměť je vyčištěna a procesor P_0 začne běžet. K aktivaci ostatních procesů může dojít pomocí instrukce FORK, kterou jakýkoliv procesor P_n může aktivovat jakýkoliv jiný procesor P_m. Procesory jsou synchronizovány na úrovni instrukcí, čili jedna instrukce každého procesoru je vždy vykonána za jednu jednotku času.

Po vykonání programu lze měřit některé vlastnosti algoritmu, například dle počtu vykonaných operací (které všechny zaberou jednu jednotku času bez ohledu na instrukce, operandy atd.) lze změřit dobu, jakou algoritmus zabral. Prostorovou

náročnost lze změřit pomocí počtu paměťových buněk, jež byly použity při běhu algoritmu.

Jelikož jsou instrukce synchronizovány, je nutné ošetřit stavy, kdy několik procesorů bude chtít pracovat zároveň s jednou paměťovou buňkou ve sdíleném paměťovém prostoru. Existuje několik typů PRAM, každý s trochu rozdílným přístupem ke konfliktům ve sdílené paměti.

Základní typy jsou [8]:

1. EREW (Exclusive-read, exclusive-write) – jedna paměťová buňka může být čtena nebo zapisována pouze jedním procesorem v jeden okamžik.
2. CREW (Concurrent-read, exclusive-write) – Libovolně mnoho procesorů může z jedné paměťové buňky číst, ale jen jeden procesor může zapisovat do sdílené paměťové buňky v jeden okamžik. Toto je výchozí varianta obecného PRAM.
3. ERCW (Exclusive-read, concurrent-write) – Pouze jeden procesor může číst z dané paměťové buňky a libovolný počet procesorů může do určité paměťové buňky zapisovat. Tento model nenachází téměř vůbec využití.
4. CRCW (Concurrent-read, concurrent-write) – Libovolný počet procesorů může zapisovat nebo číst jakoukoliv paměťovou buňku. Tento model je nejsilnější z uvedených PRAM modelů.

Pro CW (concurrent-write) modely, tedy modely, kde více procesorů může najednou zapisovat do jedné paměťové buňky, je nutné dále řešit, co se stane při konfliktu dvou a více procesorů, které zapisují různé hodnoty na dané paměťové místo. Existuje několik algoritmů [8]:

1. undefined CW – na paměťové místo je zapsána nedefinovaná (*undefined*) hodnota.
2. detecting CW – do paměťové buňky je zapsána speciální hodnota, která značí, že byla detekována kolize.
3. random CW – do buňky je zapsána hodnota, jež je náhodně vybrána ze všech hodnot poskytnutých procesory, chtějícími na dané paměťové místo zapisovat.
4. common CW – pokud je zapisovaná hodnota všech procesorů stejná, zapíše se tato hodnota.

5. max CW – do paměťové buňky se zapíše nejvyšší hodnota ze všech možností, které byly navrženy procesory, chtějícími na dané paměťové místo zapisovat.
6. min CW – podobné předchozímu modelu s tím rozdílem, že do paměťové buňky se zapíše hodnota nejmenší ze všech nabízených.
7. reduction CW – na všechny hodnoty nabízené procesory se aplikuje aritmetická či logická operace (např. AND nebo součet) a do paměťového místa je zapsán výsledek této operace.
8. priority CW – zapsána je hodnota procesoru s vyšší prioritou. Priorita může být určena například podle indexu *i* procesoru.

Kapitola 4

Aplikace

Při zvažování, kterou datovou strukturu implementovat a poté nad vybranými implementacemi provádět výkonnostní testy, jsem zvažoval několik faktorů. Jednak jsem se rozhodl implementovat strukturu, která je často využívaná a která má také vhodné uplatnění v paralelním programování. Dalším faktorem byl také počet existujících rozdílných návrhů algoritmů, které daná datová struktura poskytuje pro základní operace nad ní. Po zvážení výše popsaných faktorů padla volba na frontu. Fronta je velmi využívána jak v sekvenčním, tak paralelním prostředí. Nějakou formu implementované fronty můžeme najít například při implementaci různých bufferů (vyrovnávací paměť), datových proudů nebo rour (pipeline), jako způsobu komunikace v operačním systému.

Její velké využití v paralelních programech také samozřejmě znamená zvýšený počet vědeckých článků a prací s různými implementacemi základních operací, které fronta poskytuje – tedy enqueue (neboli push, vložení prvku do fronty) a dequeue (nebo pop či top, výběr prvku z čela fronty). Existuje tak mnoho různých neblokujících implementací. Nabízí se tak možnost zajímavého porovnání mezi neblokujícími a blokujícími implementacemi sdílené fronty.

4.1 Popis aplikace

Celá aplikace by měla fungovat jako knihovna, ze které si může programátor vybrat implementaci fronty, která mu nejvíce vyhovuje a je výkonnostně vhodná pro jeho projekt. Samozřejmě by měl programátor zohlednit také systém, na kterém aplikace poběží.

Použité technologie

C++

C++ je multiparadigmatický programovací jazyk, který byl vyvinut jako nadstavba jazyka C. Jazyk prošel několika standardy a v současné době se nachází ve standardu z roku 2011 označovaném jako C++11. Tento standard přinesl mnoho změn a vylepšení, některé z nich využívám i při implementaci mé aplikace. Je vhodné zmínit i kolekci C++ knihoven se souhrnným názvem BOOST knihovny. Jedná se o velkou a velmi dobře udržovanou kolekci knihoven s obecným a širokým využitím. Pro paralelní programování nabízí C++ knihovnu *std::atomic*, která poskytuje některé atomické operace (např. compare-and-swap) a atomický datový typ. Umožnuje také správu vláken díky knihovně *std::thread*.

Pokud chceme použít blokující zámky, existuje knihovna *std::mutex*, která poskytuje základní implementaci zámků.

Umožňuje také šablonování, což znamená, že funkce nebo třída může díky šabloně fungovat pro mnoho datových typů, aniž by musela být zapsána a upravena pro každý datový typ zvlášť.

GCC

GCC (= gnu compiler collection) je sada překladačů programovacích jazyků. Původně se jednalo o překladač pouze jazyka C, později byly přidávány další jazyky a nyní již obsahuje překladače mnoha jazyků, například Fortran, ADA nebo C++. Pro jazyky z rodiny jazyka C nabízí GCC některé atomické funkce. Není tak třeba používat robustní knihovny přímo programovacího jazyka, ale lze využít přímo překladačové implementace atomických funkcí, jako je compare-and-swap. Poskytuje také atomické operace pro práci s blokujícími synchronizačními mechanismy – zámky. Ve své práci využívám právě GCC implementaci atomické operace compare-and-swap kvůli její rychlosti a snadné použitelnosti.

Algoritmy

Pro implementaci jsem se snažil vybrat různorodé algoritmy tak, aby programátor neměl na výběr pouze velmi podobné algoritmy, ale aby si mohl najít algoritmus,

jenž co nejlépe splňuje programátorovu širokou škálu požadavků. Implementoval jsem jak blokující, tak neblokující algoritmy a také jeden semi-blokující algoritmus (což je kombinace blokujících a neblokujících operací).

4.2 Implementace

Základní stavbu knihovny tvoří třída *IQueue*, která se chová jako rozhraní (interface). Neimplementuje tedy žádné metody, pouze určuje povinné metody, které potom musí jednotlivé fronty implementovat. Obsahuje také základní definici fronty jako lineárního seznamu – ukazatel na prvky na začátek (čelo) fronty a ukazatel na konec fronty. Samotné propojení prvků pak již zajišťují jednotlivé uzly seznamu (prvky fronty) svými ukazateli na následující prvek. Základní prvek je zobrazen v pseudokódu 4.1. Celá aplikace je šablonována – fronty je možné využít na libovolný datový typ či třídu. V ukázkách kódu je tento libovolný šablonový datový typ označen jako *T*.
Pseudokód tříd front je zobrazen v 4.2.

```
struct node  // obecný uzel
{
    T data;     // data prvku
    node *next; // ukazatel na další prvek
}
```

Pseudokód 4.1: Prvek fronty

```
class IQueue  // interface class
{
    public :
        void push(T n);  // vložení nového prvku do fronty
        T pop();         // výběr prvku
        string name();   // jméno fronty

    protected:
        // definice ukazatelů na začátek a konec fronty
        node *front = nullptr;
        node *tail = nullptr;
};
```

Pseudokód 4.2: Rozhraní fronty

Fronta s jedním zámkem

Název třídy: LockQ

Tato fronta představuje základní a zcela naivní přístup k synchronizaci fronty. Pracuje pouze s jedním zámkem z C++ knihovny *std::mutex*, který zamyká kritickou sekci jak při vkládání prvku do fronty, tak při vybírání prvku z čela fronty. Jedná se tak o blokující přístup. Algoritmus je stejný jako pro sekvenční frontu, pouze s přidáním zámku pro zajištění konzistence. Při vkládání prvku do fronty se nejprve kontroluje, zda je fronta prázdná, pokud ano, ukazatele na konec a začátek fronty jsou nastaveny na nově vkládaný první prvek. Pokud je neprázdná, nastaví se ukazatel na následující prvek posledního prvku na nově vkládaný prvek a posune se na něj i ukazatel na konec fronty.

Při vybírání prvku z fronty se ukazatel na čelo fronty akorát posune na prvek následující. Pokud je ve frontě pouze jeden prvek, oba ukazatele – na čelo a na konec fronty - se po výběru prvku nastaví na inicializační hodnotu *null*.

Fronta s dvěma zámky

Název třídy: TwoLockQ

Jedná se o další blokující algoritmus, tentokrát však využívající dva zámky místo jednoho. Tento algoritmus byl představen M. Michaelem a M. L. Scottem [9]. Jeden zámek je použit pro operaci vkládání prvku do fronty a jiný zámek potom při vybírání prvku z fronty. To umožnuje paralelní provádění obou operací. Při inicializaci je nutné vložit do čela fronty „dummy“ (pomocný) prvek. Tento prvek pak zaručuje, že při operaci vkládání nikdy není třeba přistupovat a modifikovat ukazatel čela fronty a při operaci vybírání prvku z čela fronty nikdy nepotřebujeme přistupovat a modifikovat ukazatel konce fronty. Takto je umožněno použít dva zámky a současně se vyhnout problémům, jako je možnost vzniku uváznutí apod.

Semi-blokující fronta

Název třídy: SemiBlockQ

Semi-blokující algoritmus znamená, že se u něj kombinují oba přístupy - část algoritmu je blokující – pracuje se zámky a další část algoritmu je neblokující. Tato

implementace používá neblokující operaci vložení prvku do fronty. Při odebírání prvku z fronty se pak využije zámků. Kvůli kombinování obou přístupů využívá fronta dvě interní datové struktury. Jednu pro vkládání položek – zásobník a druhou pro výběr položek - frontu.

Vkládání vždy probíhá do zásobníku pro vkládání prvků. Operace vkládání vyžaduje pouze jeden compare-and-swap, a to díky použití právě zásobníku – prvek se vkládá na začátek a ne na konec.

Při operaci výběru prvku se zase pokusíme vybrat prvek z čela fronty pro výběr položek. Pokud je tato fronta prázdná, je nutno ji naplnit prvky ze zásobníku pro vkládání. Toto se provede jednoduchým cyklením nad celým zásobníkem a v podstatě se položky jdoucí ze zásobníku do fronty převrátí v pořadí (tak, aby poslední prvek zásobníku byl první prvek fronty [14].

Michael-Scott fronta

Název třídy: MSQ

Jedná se o neblokující frontu [9], která bývá zřejmě nejčastější volbou pro neblokující implementaci fronty. Fronta vyžaduje na začátku opět existenci „dummy“ prvku. Tento prvek se nachází na začátku fronty a vždy na něj ukazuje ukazatel na čelo fronty. Obě operace (vkládání/vybírání) používají několikrát compare-and-swap. Vždy je zkontrolováno, zda během běhu algoritmu nedošlo již k odstranění prvku, se kterým pracujeme. Pokud se nezměnil ukazatel na konec fronty, atomicky zkusíme vložit prvek do seznamu a poté také atomicky, vždy pomocí compare-and-swap na něj přesunout ukazatel na konec fronty. Obdobně při vybírání se atomicky pokusíme nastavit ukazatel na čelo fronty na další prvek a pokud operace neuspěje, znamená to, že fronta již byla modifikována jinde a proces se pokusí o výběr prvku znovu.

Valois fronta

Název třídy: ValoisQ

Starší neblokující fronta [10] opět využívající „dummy“ prvek. Ukazatel na čelo fronty při vybírání prvku z fronty ukazuje na prvek, který byl naposledy vybrán –

tedy zastupuje „dummy“ prvek. Použití dummy prvku nám umožňuje vyhnout se problémům, pokud je fronta prázdná či má jen jeden prvek. Také zabraňuje konfliktům při vkládání a vybírání prvků při situaci, kdy se ve frontě nachází jen jeden prvek. Pro operaci vkládání jsem implementoval variantu, jež počítá s tím, že ukazatel na konec fronty se často nachází blízko konce, ne však vždy až na úplném konci (jiný proces mezitím vložil prvek). Nejprve se tedy cyklí pomocí ukazatelů v lineárním seznamu na konec a poté se atomicky vloží prvek a nakonec se aktualizuje ukazatel na konec fronty.

Ladan-Mozes fronta

Název třídy: MozesQ

Poměrně nová neblokující implementace fronty [11] snažící se řešit nedostatky předchozích dvou neblokujících front – velký počet nutných provedení (a uspění) operací compare-and-swap. Opět je založena na myšlence vkládání prvků na začátek datové struktury, podobně jako semi-blokující fronta. Fronta je zde implementovaná jako obousměrný lineární seznam. Prvek seznamu je v pseudokódu 4.3. Vkládání na začátek vyžaduje k implementaci pouze jednu compare-and-swap operaci. Při inicializaci je do fronty opět vložen „dummy“ prvek, během používání fronty pak ukazatel na konec fronty vždy ukazuje na poslední přidaný prvek a ukazatel na čelo fronty pak na nejstarší – nejpozději přidaný prvek. Při vyprázdnění fronty opět oba ukazatele budou ukazovat na „dummy“ prvek.

Využívání dvojsměrného seznamu může vést k problémům. Ukazatele na další prvek se mění pouze při vkládání prvku a tento průchod – tedy od čela fronty až po konec za pomocí ukazatelů na další prvek – je vždy garantován jako správný. Ukazatele na předchozí prvek jsou však nastavovány při operaci vkládání „optimisticky“ – probíhá až po atomické operaci compare-and-swap, která vkládá prvek do fronty a nastavuje ukazatel na konec fronty. U těchto dvou operací však není garantováno, že budou provedeny atomicky, pouze se to předpokládá. Jelikož však máme garantovanou správnost alespoň pro průchod přes ukazatele na následující prvek, je možné pouze správnost opačného průchodu pouze

předpokládat a nevěnovat zatím pozornost tomu, zda výše popsané dvě operace proběhly atomicky a správně. Pozornost tomu bude věnována, až to bude opravdu nutné. Správnost ukazatelů na předchozí prvek se pak testuje až při vybírání prvku z fronty. Pokud se najde kolize mezi ukazateli, volá se funkce *fixList*, která projde celou frontu pomocí ukazatelů na následující prvek, a nastaví aktuální ukazatele na předchozí prvek u všech prvků fronty.

```
struct mnode  // uzel používaný v MozesQ
{
    T data;      // data uzlu
    mnode *next; // ukazatel na další prvek
    mnode *prev; // ukazatel na předchozí prvek
};
```

Pseudokód 4.3: Prvek Ladan-Mozes fronty

4.3 Měření výkonnosti algoritmů

Tato práce má být zaměřena na porovnání výkonnosti různých algoritmů, zvláště pak porovnání bezzámkových a zámkových variant. Pro frontu byla implementována široká škála algoritmům jak bezzámkových, tak algoritmů využívající zámky. Jelikož různé algoritmy mohou benefitovat z různých faktorů, implementované algoritmy byly testovány na různých strojích s rozdílnými specifikacemi. Všechny implementace byly podrobeny stejnému testu, který spočíval v neustálém provádění operací push (neboli enqueue - vložení prvku do fronty) a pop (neboli dequeue - vybrat prvek z fronty). V testech byl také postupně zvyšován počet vláken pracujících s datovou strukturou, aby se odhalily případné nedostatky algoritmu při velkém nebo naopak malém počtu paralelně běžících vláken. Všechny testy na všech strojích byly spuštěny několikrát a v tabulkách a grafech je prezentovány výsledky, které odpovídají vždy průměru ze všech naměřených odpovídajících hodnot.

Fronty v testech jsou používány s datovým typem *int* a čísla vkládaná do fronty jsou generována náhodně pomocí C++ funkce *rand()*.

Testovací zařízení

Pro testování výkonnosti front je vhodné testovat na různých typech strojů s dost odlišnými specifikacemi. Tak se můžou nejlépe odhalit případné výchylky či nedostatky v algoritmu.

Testovací zařízení č. 1 bylo zvoleno jako zástupce uživatelských počítačů. Jedná se o notebook Lenovo Thinkpad, s dvoujádrovým procesorem Intel Core i3-380UM, 3MB cache, 4GB RAM.

Testovací zařízení č. 2 byl výkonný multiprocesorový systém používaný na vědecké výpočty. Je osazen osmi výkonnými procesory Intel Core i7-3770K CPU @ 3.50GHz, každý obsahuje 4 jádra, 8MB cache, 32GB RAM.

Testy

Jak již bylo popsáno výše, fronty byly testovány při různých podmínkách.

Test č. 1 – fronta byla naplněna vždy před začátkem testu na 5 prvků. Poté bylo spuštěno 50 000 testovacích funkcí (pseudokód 4.4) jedním vláknem – jedna testovací funkce obsahuje vždy jedno volání enqueue (vložení prvku) a jedno volání dequeue (vyjmutí prvku). Pro další test jedné fronty se vždy použila fronta nová. Testy začaly s jedním běžícím vláknem, po dokončení jedné sady testů se vždy jedno vlákno přidalo pro další sadu testů. V poslední sadě testů běželo současně 16 vláken. V poslední sadě testů byly tedy fronty podrobeny 800 000 (50 000*16) volání testovacích funkcí. Kód testu lze nalézt v pseudokódu 4.5.

Test č. 2 – stejný jako předchozí test, pouze s tím rozdílem, že fronty byly naplněny před začátkem testu na 1000 prvků

Testovacímu programu lze nastavit různé parametry, výčet těchto parametrů je uveden v tabulce 4.1

Parametr	Hodnota	Význam
-q	řetězec – {“lock“, “mozes“, “ms“, “semiblock“, “twolock“, “valois“}	Určuje testovanou frontu
-i	číslo - int	Počet prvků ve frontě
-t	číslo - int	Maximální počet běžících vláken

Tabulka 4.1: Parametry testovacího programu

```
        for(int i = 0; i < LOOPS; ++i)
        {
            if(i % 2)             // polovina operací je vybírání prvku
                queue->pop();
            else                  // polovina vkládaní
                queue->push(rand());
        }
```

Pseudokód 4.4: Testovací funkce

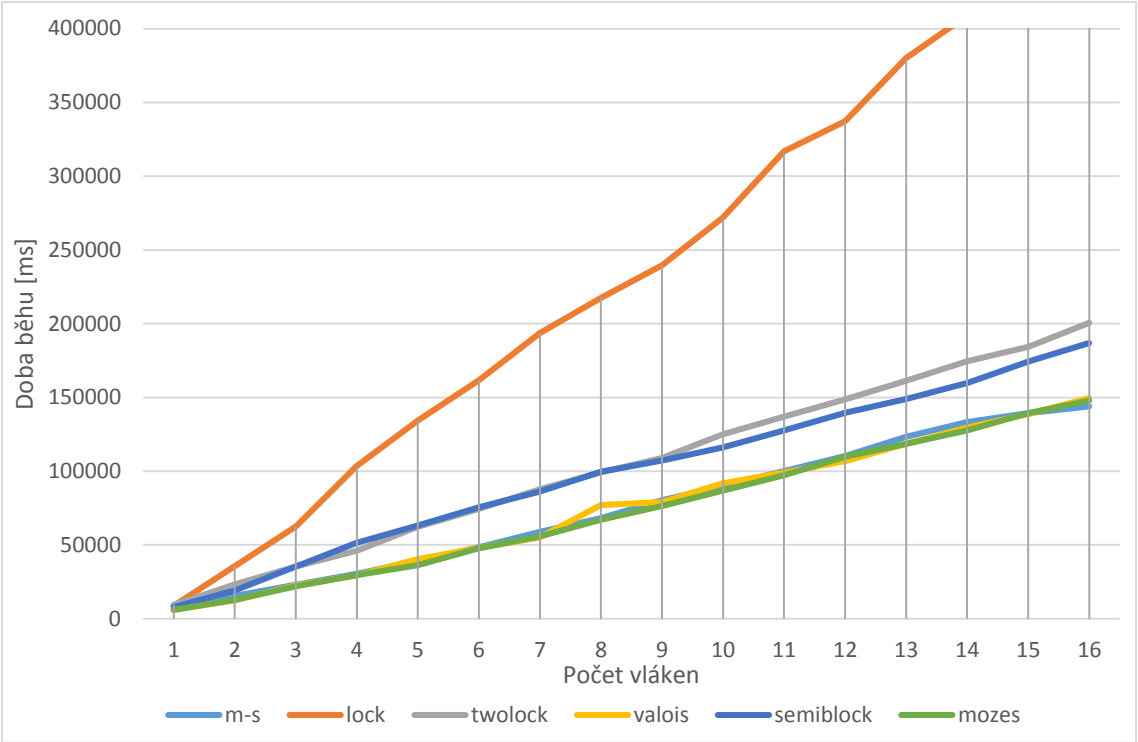
```
for(int threadsCount = 1; threadsCount <= max_threads; ++threadsCount)
{   // testuje od 1 vlákna až po požadovaný počet max

        for(int i = 0; i < initial; ++i)
            queue->push(rand());       // naplníme frontu požadovaným počtem prvků


        int start_time = get_time();    // začneme měřit čas

        // spustíme vlákna
        for (int i = 0; i < threadsCount; i++)
            // vlákno začne vykonávat testovací funkci
            start_thread( test_function );

        threads_join();     // počkáme na dokončení všech vláken

        int end_time = get_time();   // přestaneme měřit čas

        int test_time = end_time - start_time;  // výsledek jako rozdíl časů

}
```

Pseudokód 4.5: Schéma testu

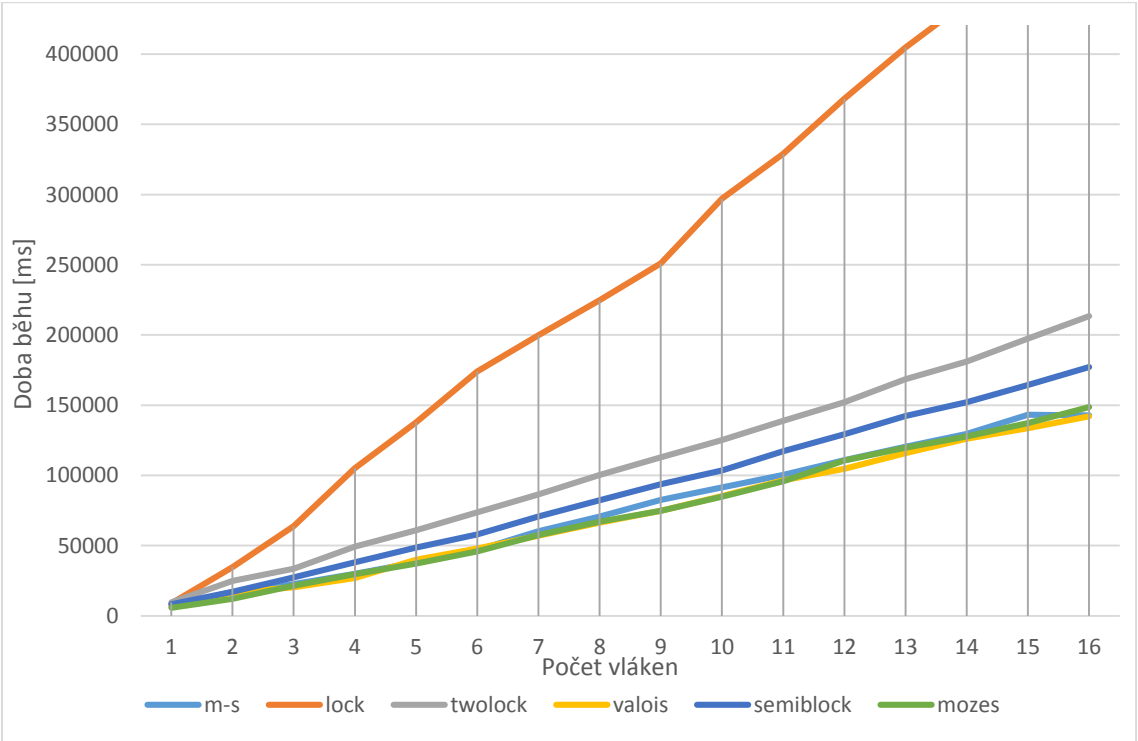
Výsledky testů

Výsledky testů lze vidět v grafické podobě v grafech 4.1 až 4.4. Testy dopadly obecně dle očekávání – nejhůře dopadly blokující algoritmy, nejlépe neblokující.
U obou strojů je až zarážející sledovat, jak velký propad je mezi frontou s jedním zámkem a všemi ostatními algoritmy. Počty prvků prakticky neměly vliv, u uživatelského PC se mírně prodloužila doba běhu blokujících front. Jelikož se jedná o nepatrný rozdíl, nelze ani vyloučit, že se jedná o statistickou odchylku.
U uživatelského PC běžely všechny neblokující algoritmy takřka stejnou dobu, u výkonného PC si vedla o něco lépe než další dva Michael-Scott fronta a potvrdila tak, že je oprávněně nejčastěji implementovanou neblokující frontou.
Pokud nemůžeme použít neblokující algoritmy, je vidět, že i fronta s dvěma zámky podává obstojný výkon, i když v porovnání s neblokujícími algoritmy stále výrazně zaostává, zvláště při vyšším počtu vláken.

Testovací zařízení č. 1 – uživatelské PC, 5 a 1000 prvků.

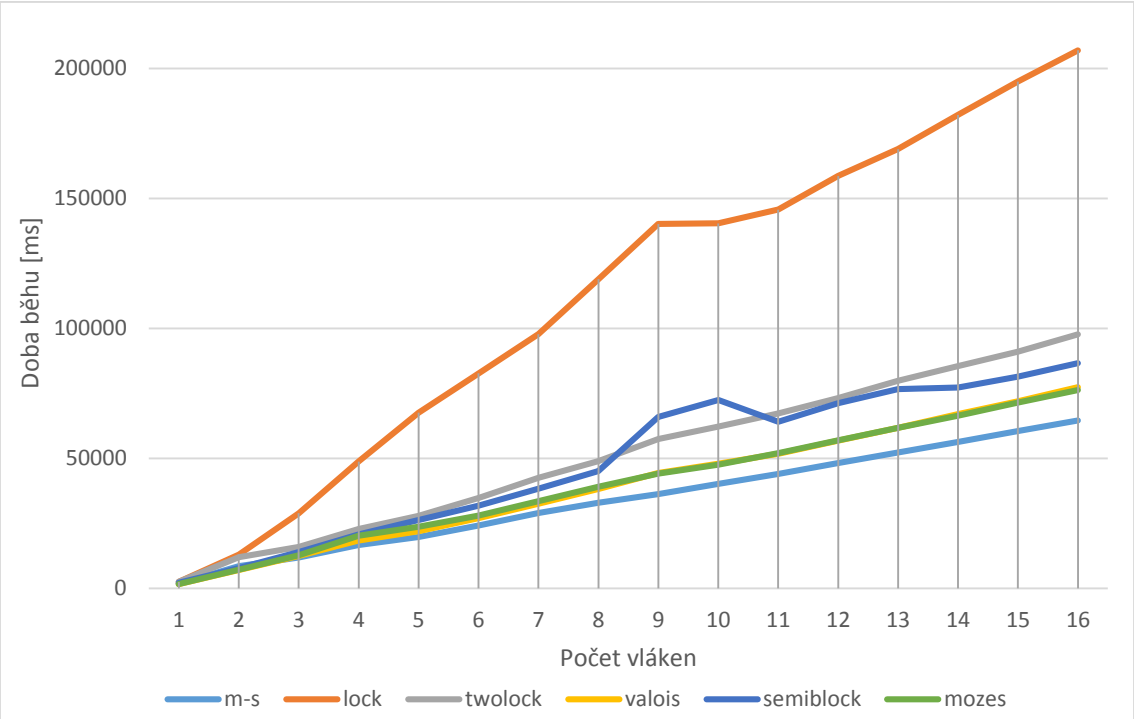


Graf 4.1: Uživatelské PC, 5 prvků

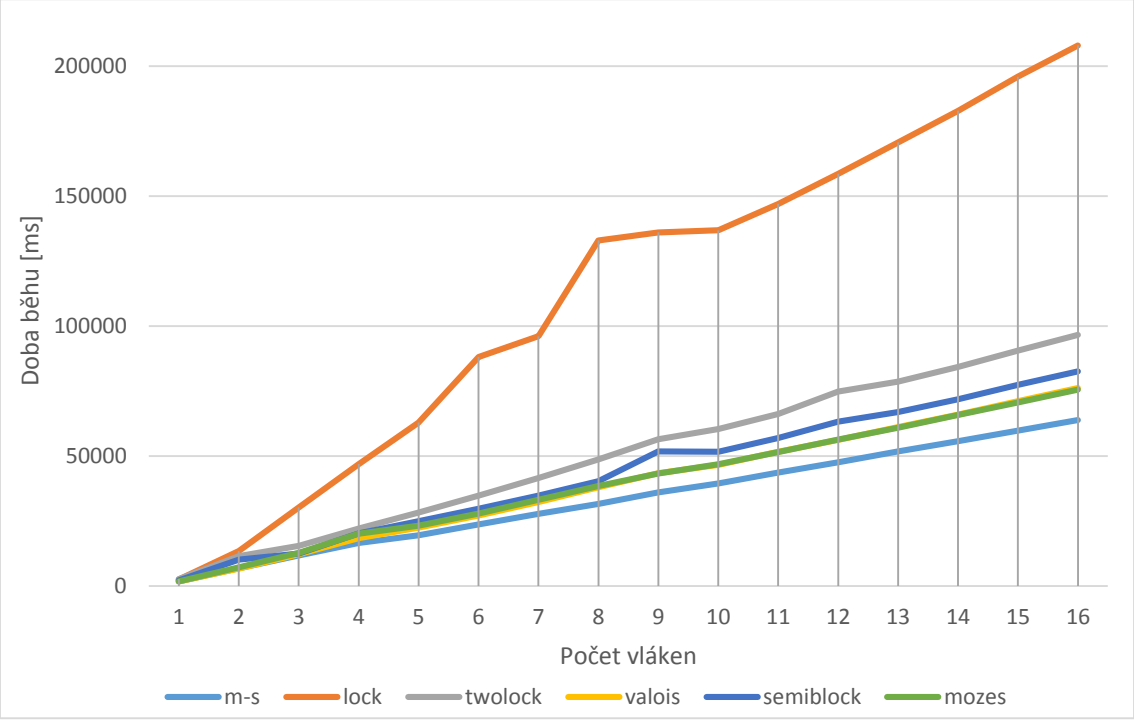


Graf 4.2: Uživatelské PC, 1000 prvků

Testovací zařízení č. 2 – výkonný počítač, 5 a 1000 prvků.



Graf 4.3: Výkonné PC, 5 prvků



Graf 4.4: Výkonné PC, 1000 prvků

Kapitola 5

Závěr

Jelikož cílem mé práce bylo naprogramovat datovou strukturu pro paralelní použití, je první část mé práce věnována teorii, která je nutná k pochopení fungování paralelních systémů a synchronizace mezi procesy a jsou také představeny základní datové struktury a jejich použití.

Jako paralelní datovou strukturu, která bude implementována, jsem vybral frontu díky jejímu dobrému využití v paralelním programování. Samotná paralelní fronta je pak implementována v šesti různých variantách. Lze zde nalézt blokující, neblokující i semi-blokující implementace. Každá varianta je implementována jako samostatná třída, kolekce těchto tříd pak může tvořit knihovnu, kterou lze využít při paralelním programování, pokud programátor potřebuje pracovat s frontou. Oddělené třídy také umožňují jednoduchou rozšiřitelnost, není problém v budoucnu doplnit další implementace paralelní fronty.

K této knihovně jsem napsal i jednoduchý test výkonnosti a všechny implementované varianty byly otestovány na výkonnost. Z výkonnostních testů je jasně vidět, jak důležité je zvolit správný algoritmus pro práci s paralelní datovou strukturou.

Jelikož i do budoucna se dá očekávat vývoj procesorů zaměřený na zvyšování počtu jader, dá se předpokládat, že paralelní programování bude také nabývat na důležitosti. Má práce může sloužit jako dobrý prostředek pro seznámení se se základy nutnými pro paralelní programování a implementovanou paralelní frontu lze přímo využít v programech pro paralelní systémy.

Literatura

[1] Scott, Michael Lee. Shared-memory synchronization. San Rafael, Calif.: Morgan & Claypool, 2013.

[2] Papadimitriou, Christos H. Computational complexity. Reading, Mass.: Addison-Wesley, 1994.

[3] Raynal, M.. Concurrent programming algorithms, principles, and foundations. Heidelberg: Springer-Verlag, 2013.

[4] Cederman, Daniel, et al. Lock-free concurrent data structures. arXiv preprint arXiv:1302.2757, 2013.

[5] Sundell, Håkan. Efficient and practical non-blocking data structures. Chalmers University of Technology, 2004.

[6] Dijkstra, Edsger Wybe. Cooperating sequential processes. Prelim. version. ed. Eindhoven, Netherlands: Technological University Eindhoven. 1965.

[7] Herlihy, Maurice P., and Jeannette M. Wing. Axioms for concurrent objects. Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. ACM, 1987.

[8] Parhami, Behrooz. Introduction to parallel processing: algorithms and architectures. Vol. 1. Springer, 1999.

[9] Michael, Maged M., and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing. ACM, 1996.

[10] Valois, John D. Implementing lock-free queues. Proceedings of the seventh international conference on Parallel and Distributed Computing Systems, 1994.

[11] Ladan-Mozes, Edya, and Nir Shavit. An optimistic approach to lock-free FIFO queues. Springer Berlin Heidelberg, 2004.

[12] Herlihy, Maurice. "Wait-free synchronization." ACM Transactions on Programming Languages and Systems (TOPLAS) 13.1 (1991): 124-149.

[13] Lentaris, George. Models of Parallel Computation and Parallel Complexity. Diss. University of Athens, 2010.

[14] Concurrent queue in C [online]. 2012-09-11. [cit. 2014-05-17]. Dostupné na URL: <https://idea.popcount.org/2012-09-11-concurrent-queue-in-c/>

Seznam příloh

Příloha 1. CD se zdrojovými kódy